WORDWARE GAME MATH LIBRARY

Learn about the key mathematical principles used in 3D graphics, games, simulations, and computational geometry.

Put mathematical theory into practice with working C++ classes, each tailored to specific geometric tasks.

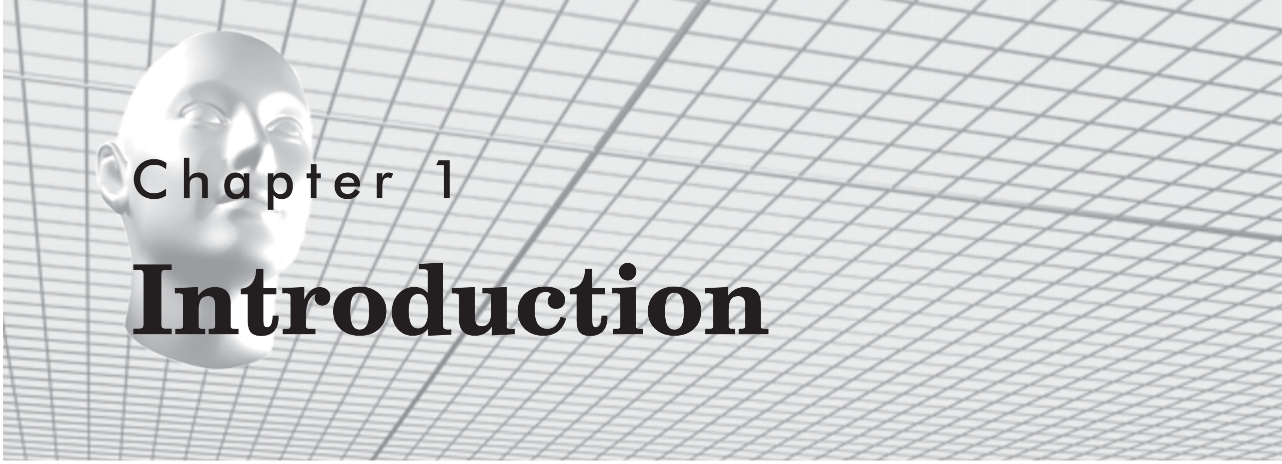
Review real-time rendering techniques with a focus on the key mathematical concepts.

Download example code and interactive demos from the gamemath.com web site.

3D Math Primer for Graphics and Game Development

Fletcher Dunn and Ian Parberry

Wordware Publishing, Inc.



Chapter 1
Introduction

1.1 What is 3D Math?

This book is about 3D math, the study of the mathematics behind the geometry of a 3D world. 3D math is related to computational geometry, which deals with solving geometric problems algorithmically. 3D math and computational geometry have applications in a wide variety of fields that use computers to model or reason about the world in 3D, such as graphics, games, simulation, robotics, virtual reality, and cinematography.

This book covers theory and practice in C++. The "theory" part is an explanation of the relationship between math and geometry in 3D. It also serves as a handy reference for techniques and equations. The "practice" part illustrates how these concepts can be applied in code. The programming language used is C++, but in principle, the theoretical techniques from this book can be applied in any programming language.

This book *is not* just about computer graphics, simulation, or even computational geometry. However, if you plan to study those subjects, you will definitely need the information in this book.

1.2 Why You Should Read This Book

If you want to learn about 3D math in order to program games or graphics, then this book is for you. There are many books out there that promise to teach you how to make a game or put cool pictures up on the screen, so why should you read *this* particular book? This book offers several unique advantages over other books about games or graphics programming:

- **A unique topic**. This book fills a gap that has been left by other books on graphics, linear algebra, simulation, and programming. It is an *introductory* book, meaning we have focused our efforts on providing thorough coverage on fundamental 3D concepts — topics that are normally glossed over in a few quick pages or relegated to an appendix in other publications (because, after all, you already know all this stuff). Our book is definitely the book you should read *first*, before buying that "Write a 3D Video Game in 21 Days" book. This book is not only an introductory book, it is also a reference book — a "toolbox" of equations and techniques that you can browse through on a first reading and then revisit when the need for a specific tool arises.

As you can see from the map, Center Street runs east-west through the middle of town. All other east-west streets (parallel to Center Street) are named based on whether they are north or south of Center Street and how far away they are from Center Street. Examples of streets which run east-west are North 3rd Street and South 15th Street.

The other streets in Cartesia run north-south. Division Street runs north-south through the middle of town. All other north-south streets (parallel to Division Street) are named based on whether they are east or west of Division street, and how far away they are from Division Street. So we have streets such as East 5th Street and West 22nd Street.

The naming convention used by the city planners of Cartesia may not be creative, but it certainly is practical. Even without looking at the map, it is easy to find the doughnut shop at North 4th and West 2nd. It's also easy to determine how far you will have to drive when traveling from one place to another. For example, to go from that doughtnut shop at North 4th and West 2nd to the police station at South 3rd and Division, you would travel seven blocks south and two blocks east.

2.2.2 Arbitrary 2D Coordinate Spaces

Before Cartesia was built, there was nothing but a large flat area of land. The city planners arbitrarily decided where the center of town would be, which direction to make the roads run, how far apart to space the roads, etc. Much like the Cartesia city planners laid down the city streets, we can establish a 2D Cartesian coordinate system anywhere we want — on a piece of paper, a chessboard, a chalkboard, a slab of concrete, or a football field.

Figure 2.5 shows a diagram of a 2D Cartesian coordinate system.

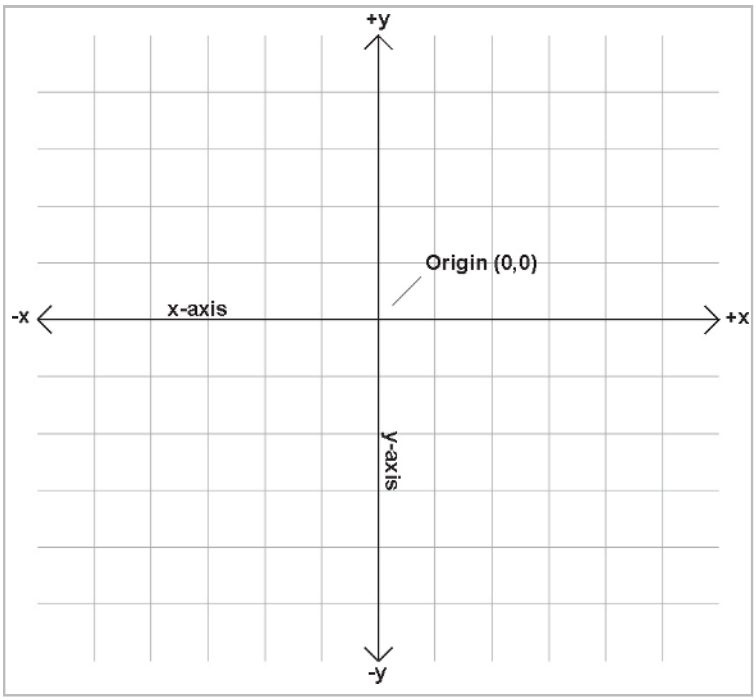


Figure 2.5: A 2D Cartesian coordinate space

As illustrated in Figure 2.5, a 2D Cartesian coordinate space is defined by two pieces of information:

- Every 2D Cartesian coordinate space has a special location, called the *origin*, which is the "center" of the coordinate system. The origin is analogous to the center of the city in Cartesia.
- Every 2D Cartesian coordinate space has two straight lines that pass through the origin. Each line is known as an *axis* and extends infinitely in two opposite directions. The two axes are perpendicular to each other. (Actually, they don't *have* to be, but most of the coordinate systems we will look at will have perpendicular axes.) The two axes are analogous to Center and Division streets in Cartesia. The grid lines in the diagram are analogous to the other streets in Cartesia.

At this point, it is important to highlight a few significant differences between Cartesia and an abstract mathematical 2D space:

- The city of Cartesia has official city limits. Land outside of the city limits is not considered part of Cartesia. A 2D coordinate space, however, extends infinitely. Even though we usually only concern ourselves with a small area within the plane defined by the coordinate space, this plane, in theory, is boundless. In addition, the roads in Cartesia only go a certain distance (perhaps to the city limits), and then they stop. Our axes and grid lines, on the other hand, each extend potentially infinitely in two directions.
- In Cartesia, the roads have thickness. Lines in an abstract coordinate space have location and (possibly infinite) length, but no real thickness.
- In Cartesia, you can only drive on the roads. In an abstract coordinate space, *every* point in the plane of the coordinate space is part of the coordinate space, not just the area on the "roads." The grid lines are only drawn for reference.

In Figure 2.5, the horizontal axis is called the x-axis, with positive x pointing to the right. The vertical axis is the y-axis, with positive y pointing up. This is the customary orientation for the axes in a diagram. Note that "horizontal" and "vertical" are terms that are inappropriate for many 2D spaces that arise in practice. For example, imagine the coordinate space on top of a desk — both axes are "horizontal," and neither axis is really "vertical."

The city planners of Cartesia could have made Center Street run north-south instead of east-west. Or they could have placed it at a completely arbitrary angle. (For example, Long Island, New York is reminiscent of Cartesia, where for convenience the streets numbered "1st Street," "2nd Street," etc., run across the island and the avenues numbered "1st Avenue," "2nd Avenue," etc., run along its long axis. The geographic orientation of the long axis of the island is an arbitrary freak of nature.) In the same way, we are free to place our axes in any way that is convenient to us. We must also decide for each axis which direction we consider to be positive. For example, when working with images on a computer screen, it is customary to use the coordinate system shown in Figure 2.6. Notice that the origin is in the upper left-hand corner, $+x$ points to the right, and $+y$ points *down* rather than up.

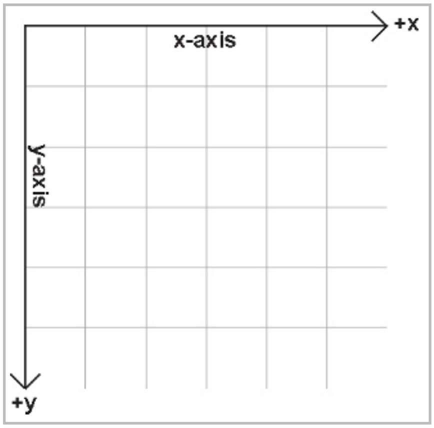


Figure 2.6: Screen coordinate space

Unfortunately, when Cartesia was being laid out, the only mapmakers were in the neighboring town of Dyslexia. The minor-level functionary who sent the contract out to bid neglected to take into account that the dyslexic mapmaker was equally likely to draw his maps with north pointing up, down, left, or right; although he always drew the east-west line at right angles to the north-south line, he often got east and west backward. When his boss realized that the job had gone to the lowest bidder, who happened to live in Dyslexia, many hours were spent in committee meetings trying to figure out what to do. The paperwork had been done, the purchase order had been issued, and bureaucracies being what they are, it would be too expensive and time-consuming to cancel the order. Still, nobody had any idea what the mapmaker would deliver. A committee was hastily formed.

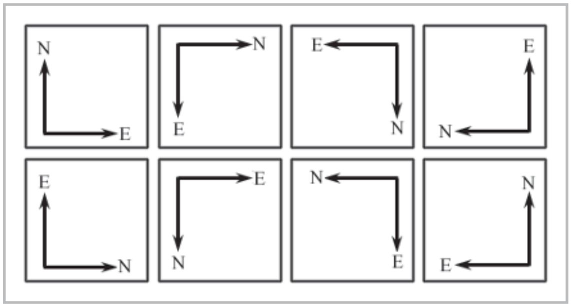


Figure 2.7: Possible map axis orientations in 2D

The committee quickly decided that there were only eight possible orientations that the mapmaker could deliver, shown in Figure 2.7. In the best of all possible worlds, he would deliver a map oriented as shown in the top-left rectangle, with north pointing to the top of the page and east to the right, which is what people usually expect. A subcommittee decided to name this the *normal orientation*.

After the meeting had lasted a few hours and tempers were beginning to fray, it was decided that the other three variants shown in the top row of Figure 2.7 were probably acceptable too, because they could be transformed to the normal orientation by placing a pin in the center of the page and rotating the map around the pin. (You can do this too by placing this book flat on a table

Chapter 8

Matrices and Linear Transformations

This chapter discusses the implementation of linear transformations using matrices. It is divided into eight sections.

- Section 8.1 describes the relationship between transforming an object and transforming the coordinate space used to describe the object.
- Sections 8.2 through 8.6 describe the primitive linear transformations of rotation, scaling, orthographic projection, reflection, and shearing, respectively. For each transformation, examples and equations are given in 2D and 3D.
- Section 8.7 shows how a sequence of primitive transformations may be combined using matrix multiplication to form a more complicated transformation.
- Section 8.8 discusses various interesting categories of transformations, including linear, affine, invertible, angle-preserving, orthogonal, and rigid body transforms.

In Chapter 7, we investigated some of the basic mathematical properties of matrices. We also developed a geometric understanding of matrices and their relationship to coordinate space transformations in general. This chapter discusses this relationship between matrices and linear transformations in more detail.

To be more specific, this chapter is concerned with expressing linear transformations in 3D using 3×3 matrices. Linear transformations were introduced in Section 7.2. Recall that one important property of *linear* transformations is that they do not contain translation. A transformation that contains translation is known as an *affine* transformation. Affine transformations in 3D cannot be implemented using 3×3 matrices. We will see a formal definition of affine transformations in Section 8.8.2, and we will learn how to use 4×4 matrices to represent affine transformations in Section 9.4.3.

8.1 Transforming an Object vs. Transforming the Coordinate Space

Before we can talk about transformations, we must be very precise about exactly *what* we are transforming. We talked briefly about the relationship between transforming objects and transforming coordinate spaces in Section 3.5. Let's take a closer look now.

Consider the 2D example of "rotating an *object* clockwise 20°." When *transforming an object*, we mean that we are transforming the points of the object. These points move to a new position, and we use the same coordinate system to express the location of the points before and after transformation. Figure 8.1 illustrates this graphically.

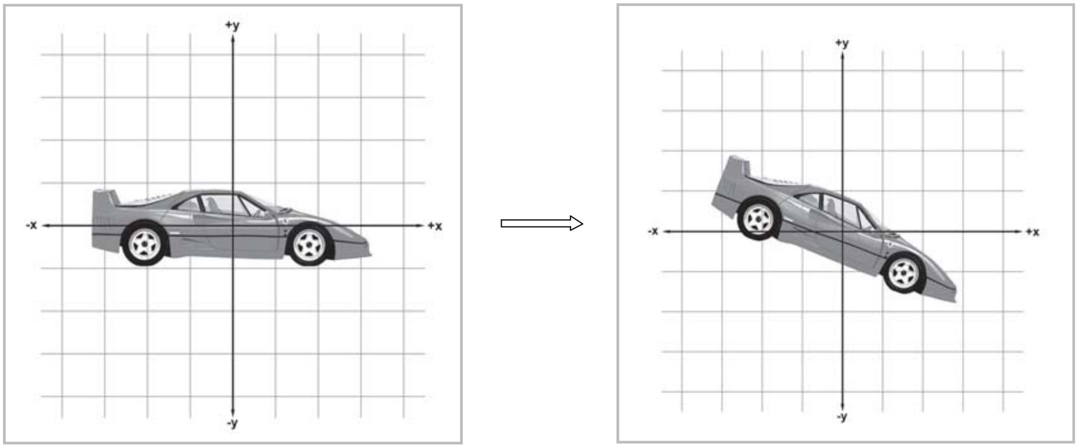


Figure 8.1: Rotating an object clockwise 20°

Compare that with the concept of *transforming the coordinate space*. When we rotate the coordinate space, the points of the object do not actually move; we are just expressing their location using a different coordinate space, as illustrated in Figure 8.2.

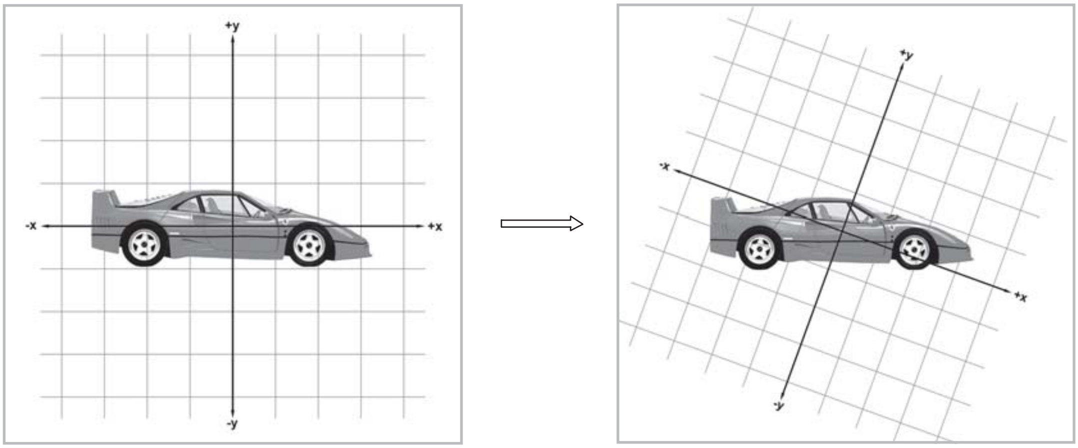


Figure 8.2: Rotating a coordinate space clockwise 20°

In a few minutes, we will show how the two types of transformations are, in a sense, equivalent. For now, let's discuss the conceptual merits of each.

The utility of transforming the object is fairly obvious. For example, in order to render the car, it will be necessary to transform the points from the object space of the car into world space, and then into camera space.

But why would we ever transform the coordinate space? Looking at Figure 8.3, it doesn't seem like there are many interesting things that we can accomplish by rotating the coordinate space into this awkward position. However, by examining Figure 8.3, we see how rotating the coordinate space can be put to good use.

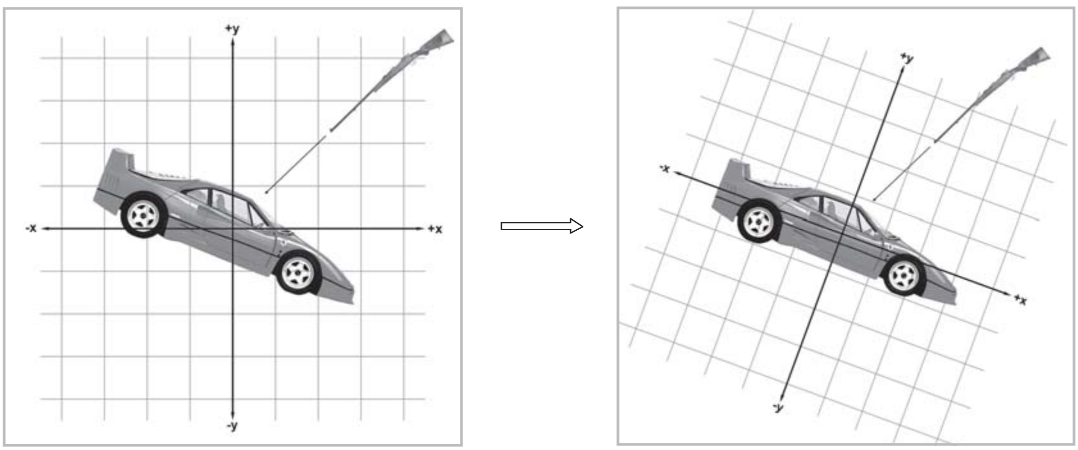


Figure 8.3: A useful example of rotating a coordinate space

In this figure, we have introduced a rifle that is firing a bullet at the car. As indicated by the coordinate space on the left, we would normally begin by knowing about the gun and the trajectory of the bullet in world space. Now, imagine transforming the coordinate space in line with the car's object space while keeping the car, the gun, and the trajectory of the bullet still. Now we know the position of the gun and the trajectory of the bullet in the object space of the car, and we could perform intersection tests to see if and where the bullet would hit the car.

Of course, we could just as easily have transformed the points of the car into world space and performed the test in world space, but that would be much slower since the car is probably modeled using many vertices and triangles, and there is only one gun. For now, don't worry about the details of actually performing the transformations; that's what the remainder of this chapter is for. Just remember that we can transform an object, or we can transform a coordinate space. Sometimes one or the other is more appropriate.

It is useful for us to maintain a conceptual distinction and to think about transforming the object in some cases and transforming the coordinate space in other cases. However, the two operations are actually equivalent. Transforming an object by a certain amount is equivalent to transforming the coordinate space by the *opposite* amount.

For example, let's take the diagram from the right-hand side of Figure 8.2, which shows the coordinate space rotated clockwise 20°. We will rotate the entire diagram (the coordinate space *and* the car) so that the coordinate space is back to the "standard" orientation on the page. Since we are rotating the entire diagram, we are merely looking at things from a different perspective, and we are not changing the relationship between the car and the coordinate space.

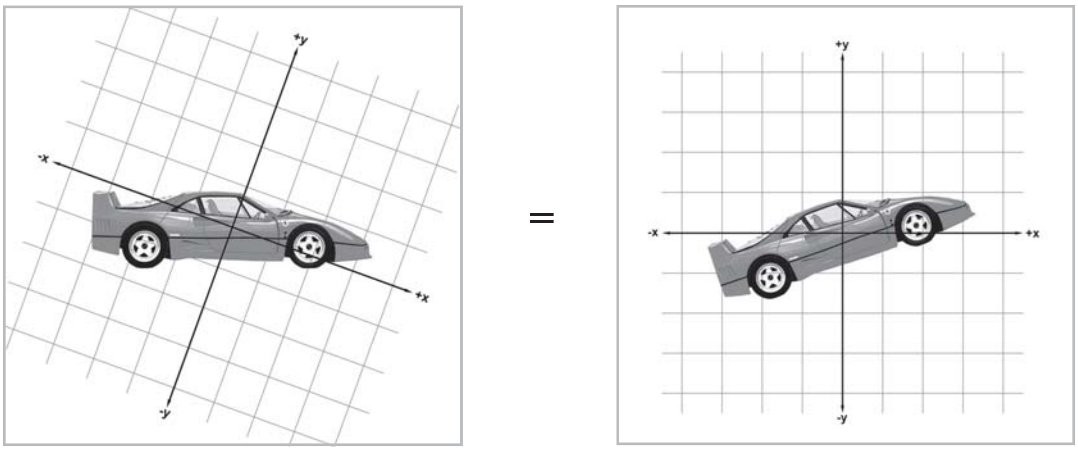


Figure 8.4: Rotating the coordinate space is the same as rotating the object by the opposite amount

Notice that this is the same as if we had started with the original diagram and rotated the car counterclockwise 20°. So rotating the coordinate space clockwise 20° is the same as rotating the object counterclockwise 20°. In general, transforming the geometry of an object is equivalent to transforming the coordinate space used to describe the geometry of the object by the exact *opposite* amount.

When multiple transformations are involved, we perform the transformations in the opposite order. For example, if we rotate the object clockwise 20° and then scale it by 200%, this is equivalent to scaling the coordinate space by 50% and then rotating the coordinate space counterclockwise 20°. We will discuss how to combine multiple transformations in Section 8.7.

The following sections present equations for constructing matrices to perform various transformations. These discussions will assume the perspective that the object is being transformed and the coordinate space remains stationary. Remember that we can always transform the coordinate space by transforming the object by the opposite amount.

8.2 Rotation

We have already seen general examples of rotation matrices. Now let's develop a more rigorous definition.

8.2.1 Rotation in 2D

In 2D, we are restricted to rotation about a point. Since we are not considering translation at the moment, we will restrict our discussion even further to rotation about the origin. A 2D rotation about the origin has only one parameter, the angle θ, which defines the amount of rotation. Counterclockwise rotation is usually (but not always) considered positive, and clockwise rotation is considered negative. Figure 8.5 shows how the basis vectors **p** and **q** are rotated about the origin, resulting in the new basis vectors **p'** and **q'**:

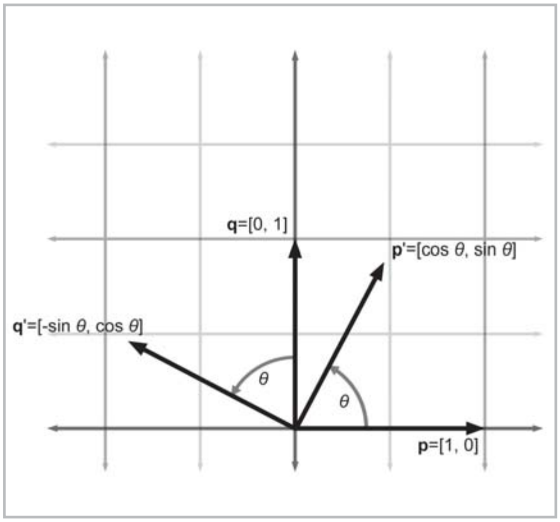


Figure 8.5: Rotation about the origin in 2D

Now that we know the values of the basis vectors after rotation, we can build our matrix:

Equation 8.1: 2D rotation matrix

$$\mathbf{R}(\theta) = \begin{bmatrix} \mathbf{p}' \\ \mathbf{q}' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

8.2.2 3D Rotation about Cardinal Axes

In 3D, rotation occurs about an axis rather than a point. (In this case, the term *axis* refers to a line about which something rotates, and it does not necessarily have to be one of the cardinal x, y, or z axes.) Again, since we are not considering translation, we will limit the discussion to rotation about an axis that passes through the origin.

When we rotate about an axis by an amout θ, we need to know which way is considered "positive" and which way is considered "negative." The standard way to do this in a left-handed coordinate system (like the coordinate system used in this book) is called the *left-hand rule*. First, we must define which way our axis points. Of course, the axis of rotation is theoretically infinite in length, but we still consider it having a positive and negative end, just like the standard cardinal axes that define our coordinate space. The left-hand rule works like this: put your left hand in the "thumbs up" position, with your thumb pointing toward the positive end of the axis of rotation. Positive rotation about the axis of rotation is in the direction that your fingers are curled. This is illustrated below:

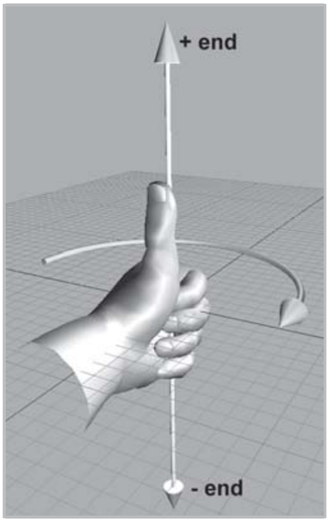


Figure 8.6: The left-hand rule defines positive rotation in a left-handed coordinate system

If you are using a right-handed coordinate system, then a similar rule applies, using your right hand instead of your left:

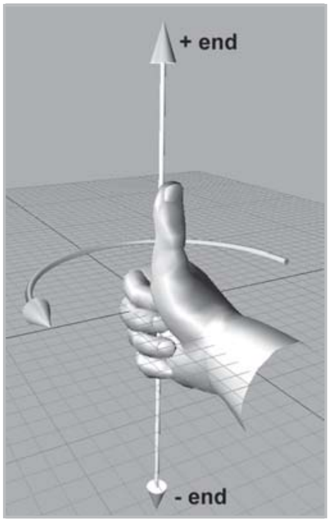


Figure 8.7: The right-hand rule defines positive rotation in a right-handed coordinate system

Figure 8.8 shows an alternative definition of positive rotation:

	Left-handed coordinate system		Right-handed coordinate system	
Viewed from	Positive rotation	Negative rotation	Positive rotation	Negative rotation
The negative end of the axis, looking toward the positive end of the axis	Counter-clockwise	Clockwise	Clockwise	Counter-clockwise
The positive end of the axis, looking toward the negative end of the axis	Clockwise	Counter-clockwise	Counter-clockwise	Clockwise

Figure 8.8: Positive and negative rotation about an axis

The most common type of rotation we will perform is a simple rotation about one of the cardinal axes. Let's start with rotation about the *x*-axis, shown in Figure 8.9:

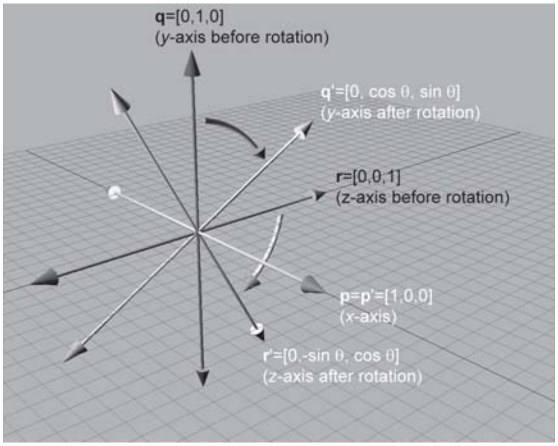


Figure 8.9: Rotating about the x-axis in 3D

After constructing a matrix from the rotated basis vectors, we have:

Equation 8.2: 3D matrix to rotate about the *x*-axis

$$\mathbf{R}_x(\theta) = \begin{bmatrix} \mathbf{p}' \\ \mathbf{q}' \\ \mathbf{r}' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$

Rotation about the *y*-axis is similar:

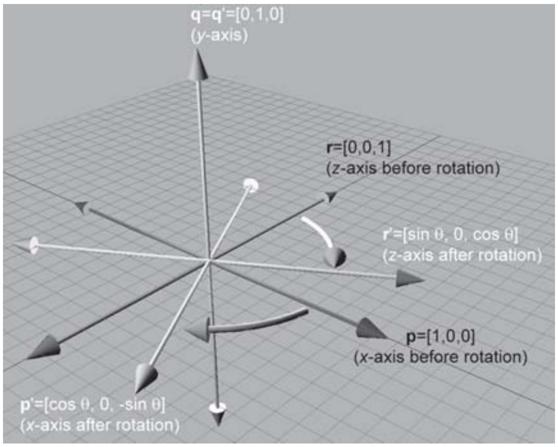


Figure 8.10: Rotating about the y-axis in 3D

The matrix to rotate about the *y*-axis:

Equation 8.3: 3D matrix to rotate about the *y*-axis

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \mathbf{p}' \\ \mathbf{q}' \\ \mathbf{r}' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

Finally, rotating about the z-axis:

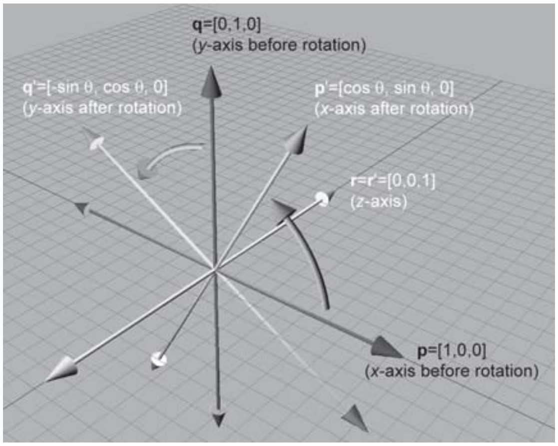


Figure 8.11: Rotating about the z-axis in 3D

Equation 8.4: 3D matrix to rotate about the z-axis

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \mathbf{p}' \\ \mathbf{q}' \\ \mathbf{r}' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

8.2.3 3D Rotation about an Arbitrary Axis

We can also rotate about an arbitrary axis in 3D, provided of course that the axis passes through the origin, since we are not considering translation at the moment. This is more complicated and less common than rotating about a cardinal axis. The axis will be defined by a unit vector **n**. As before, we will define θ to be the amount of rotation about the axis.

Let's derive a matrix to rotate about **n** by the angle θ. In other words, we wish to derive the matrix $\mathbf{R}(\mathbf{n}, \theta)$ such that

$$\mathbf{v}\mathbf{R}(\mathbf{n}, \theta) = \mathbf{v}'$$

where $\mathbf{v}'$ is the vector **v** after rotating about **n**. Let us first see if we can express $\mathbf{v}'$ in terms of **v**, **n**, and θ. The basic idea is to solve the problem in the plane perpendicular to **n**, which is a much simpler 2D problem. To do this, we will separate **v** into two values, $\mathbf{v}_{\|}$ and $\mathbf{v}_{\perp}$, which are parallel and perpendicular to **n**, respectively, such that $\mathbf{v} = \mathbf{v}_{\|} + \mathbf{v}_{\perp}$. (We learned the math for this in Section 5.10.3.) Since $\mathbf{v}_{\|}$ is parallel to **n**, it will not be affected by the rotation about **n**. So if we can rotate $\mathbf{v}_{\perp}$ about **n** to compute $\mathbf{v}'_{\perp}$, then we can compute $\mathbf{v}' = \mathbf{v}_{\|} + \mathbf{v}'_{\perp}$. To compute $\mathbf{v}'_{\perp}$, we will construct the vectors $\mathbf{v}_{\|}$, $\mathbf{v}_{\perp}$, and an intermediate vector **w**, according to Figure 8.12.

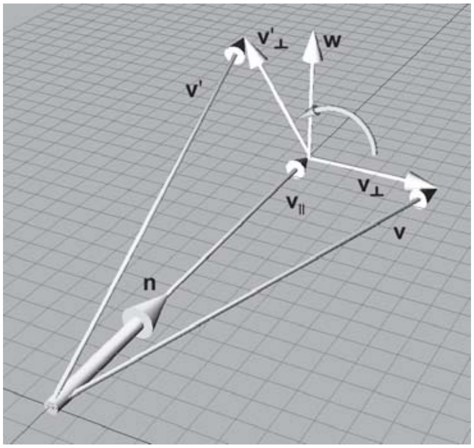


Figure 8.12: Rotating a vector about an arbitrary axis

The diagram above illustrates the following vectors:

- $\mathbf{v}_{||}$ is the portion of **v** that is parallel to **n**. Another way to say this is that $\mathbf{v}_{||}$ is the value of **v** *projected onto* **n**. This can be computed by $(\mathbf{v}\cdot\mathbf{n})\mathbf{n}$.
- $\mathbf{v}_{\perp}$ is the portion of **v** that is perpendicular to **n**. Since $\mathbf{v} = \mathbf{v}_{||} + \mathbf{v}_{\perp}$, $\mathbf{v}_{\perp}$ can be computed by $\mathbf{v} - \mathbf{v}_{||}$. $\mathbf{v}_{\perp}$ is the result of projecting **v** onto the plane perpendicular to **n**.
- **w** is a vector that is mutually perpendicular to $\mathbf{v}_{||}$ and $\mathbf{v}_{\perp}$, and it has the same length as $\mathbf{v}_{\perp}$. **w** and $\mathbf{v}_{\perp}$ lie in the plane perpendicular to **n**. **w** is the result of rotating $\mathbf{v}_{\perp}$ about **n** by 90°. **w** can be computed by $\mathbf{n}\times\mathbf{v}_{\perp}$.

Now we can see that the portion of **v'** perpendicular to **n** is given by:

$$\mathbf{v}'_{\perp} = \cos\theta\mathbf{v}_{\perp} + \sin\theta\mathbf{w}$$

Substituting for $\mathbf{v}_{\perp}$ and **w**:

$$\begin{aligned}
\mathbf{v}_{||} &= (\mathbf{v}\cdot\mathbf{n})\,\mathbf{n} \\
\mathbf{v}_{\perp} &= \mathbf{v} - \mathbf{v}_{||} \\
&= \mathbf{v} - (\mathbf{v}\cdot\mathbf{n})\,\mathbf{n} \\
\mathbf{w} &= \mathbf{n}\times\mathbf{v}_{\perp} \\
&= \mathbf{n}\times\left(\mathbf{v} - \mathbf{v}_{||}\right) \\
&= \mathbf{n}\times\mathbf{v} - \mathbf{n}\times\mathbf{v}_{||} \\
&= \mathbf{n}\times\mathbf{v} - \mathbf{0} \\
&= \mathbf{n}\times\mathbf{v} \\
\mathbf{v}'_{\perp} &= \cos\theta\mathbf{v}_{\perp} + \sin\theta\mathbf{w} \\
&= \cos\theta\,(\mathbf{v} - (\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}) + \sin\theta\,(\mathbf{n}\times\mathbf{v})
\end{aligned}$$

Substituting for **v**', we have:

$$\begin{aligned}\mathbf{v}' &= \mathbf{v}'_\perp + \mathbf{v}_\parallel \\ &= \cos\theta\,(\mathbf{v} - (\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}) + \sin\theta\,(\mathbf{n}\times\mathbf{v}) + (\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}\end{aligned}$$

Now that we have expressed **v**' in terms of **v**, **n**, and θ, we can compute what the basis vectors are after transformation and construct our matrix. We'll work through the first basis vector:

$$\begin{aligned}
\mathbf{p} &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \\
\mathbf{p}' &= \cos\theta\,(\mathbf{p} - (\mathbf{p}\cdot\mathbf{n})\,\mathbf{n}) + \sin\theta\,(\mathbf{n}\times\mathbf{p}) + (\mathbf{p}\cdot\mathbf{n})\,\mathbf{n} \\
&= \cos\theta\left(\begin{bmatrix}1\\0\\0\end{bmatrix} - \left(\begin{bmatrix}1\\0\\0\end{bmatrix}\cdot\begin{bmatrix}\mathbf{n}_x\\\mathbf{n}_y\\\mathbf{n}_z\end{bmatrix}\right)\begin{bmatrix}\mathbf{n}_x\\\mathbf{n}_y\\\mathbf{n}_z\end{bmatrix}\right) + \sin\theta\left(\begin{bmatrix}\mathbf{n}_x\\\mathbf{n}_y\\\mathbf{n}_z\end{bmatrix}\times\begin{bmatrix}1\\0\\0\end{bmatrix}\right) + \left(\begin{bmatrix}1\\0\\0\end{bmatrix}\cdot\begin{bmatrix}\mathbf{n}_x\\\mathbf{n}_y\\\mathbf{n}_z\end{bmatrix}\right)\begin{bmatrix}\mathbf{n}_x\\\mathbf{n}_y\\\mathbf{n}_z\end{bmatrix} \\
&= \cos\theta\left(\begin{bmatrix}1\\0\\0\end{bmatrix} - \mathbf{n}_x\begin{bmatrix}\mathbf{n}_x\\\mathbf{n}_y\\\mathbf{n}_z\end{bmatrix}\right) + \sin\theta\begin{bmatrix}0\\\mathbf{n}_z\\-\mathbf{n}_y\end{bmatrix} + \mathbf{n}_x\begin{bmatrix}\mathbf{n}_x\\\mathbf{n}_y\\\mathbf{n}_z\end{bmatrix} \\
&= \cos\theta\begin{bmatrix}1-\mathbf{n}_x{}^2\\-\mathbf{n}_x\mathbf{n}_y\\-\mathbf{n}_x\mathbf{n}_z\end{bmatrix} + \sin\theta\begin{bmatrix}0\\\mathbf{n}_z\\-\mathbf{n}_y\end{bmatrix} + \begin{bmatrix}\mathbf{n}_x{}^2\\\mathbf{n}_x\mathbf{n}_y\\\mathbf{n}_x\mathbf{n}_z\end{bmatrix} \\
&= \begin{bmatrix}\cos\theta - \mathbf{n}_x{}^2\cos\theta\\-\mathbf{n}_x\mathbf{n}_y\cos\theta\\-\mathbf{n}_x\mathbf{n}_z\cos\theta\end{bmatrix} + \begin{bmatrix}0\\\mathbf{n}_z\sin\theta\\-\mathbf{n}_y\sin\theta\end{bmatrix} + \begin{bmatrix}\mathbf{n}_x{}^2\\\mathbf{n}_x\mathbf{n}_y\\\mathbf{n}_x\mathbf{n}_z\end{bmatrix} \\
&= \begin{bmatrix}\cos\theta - \cos\theta\mathbf{n}_x{}^2 + \mathbf{n}_x{}^2\\-\mathbf{n}_x\mathbf{n}_y\cos\theta + \mathbf{n}_z\sin\theta + \mathbf{n}_x\mathbf{n}_y\\-\mathbf{n}_x\mathbf{n}_z\cos\theta - \mathbf{n}_y\sin\theta + \mathbf{n}_x\mathbf{n}_z\end{bmatrix} \\
&= \begin{bmatrix}\mathbf{n}_x{}^2(1-\cos\theta) + \cos\theta\\\mathbf{n}_x\mathbf{n}_y(1-\cos\theta) + \mathbf{n}_z\sin\theta\\\mathbf{n}_x\mathbf{n}_z(1-\cos\theta) - \mathbf{n}_y\sin\theta\end{bmatrix}
\end{aligned}$$

The derivation of the other two basis vectors is similar and produces the following results:

$$\begin{aligned}\mathbf{q} &= \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \\ \mathbf{q}' &= \begin{bmatrix}\mathbf{n}_x\mathbf{n}_y(1-\cos\theta) - \mathbf{n}_z\sin\theta\\\mathbf{n}_y{}^2(1-\cos\theta) + \cos\theta\\\mathbf{n}_y\mathbf{n}_z(1-\cos\theta) + \mathbf{n}_x\sin\theta\end{bmatrix}\end{aligned} \qquad \begin{aligned}\mathbf{r} &= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \\ \mathbf{r}' &= \begin{bmatrix}\mathbf{n}_x\mathbf{n}_z(1-\cos\theta) + \mathbf{n}_y\sin\theta\\\mathbf{n}_y\mathbf{n}_z(1-\cos\theta) - \mathbf{n}_x\sin\theta\\\mathbf{n}_z{}^2(1-\cos\theta) + \cos\theta\end{bmatrix}\end{aligned}$$

Note: We used column vectors above strictly so that the equations would format nicely.

Constructing the matrix from these basis vectors:

Equation 8.5: 3D matrix to rotate about an arbitrary axis

$$\mathbf{R}(\mathbf{n},\theta) = \begin{bmatrix}\mathbf{p}'\\\mathbf{q}'\\\mathbf{r}'\end{bmatrix} = \begin{bmatrix}\mathbf{n}_x{}^2(1-\cos\theta)+\cos\theta & \mathbf{n}_x\mathbf{n}_y(1-\cos\theta)+\mathbf{n}_z\sin\theta & \mathbf{n}_x\mathbf{n}_z(1-\cos\theta)-\mathbf{n}_y\sin\theta\\\mathbf{n}_x\mathbf{n}_y(1-\cos\theta)-\mathbf{n}_z\sin\theta & \mathbf{n}_y{}^2(1-\cos\theta)+\cos\theta & \mathbf{n}_y\mathbf{n}_z(1-\cos\theta)+\mathbf{n}_x\sin\theta\\\mathbf{n}_x\mathbf{n}_z(1-\cos\theta)+\mathbf{n}_y\sin\theta & \mathbf{n}_y\mathbf{n}_z(1-\cos\theta)-\mathbf{n}_x\sin\theta & \mathbf{n}_z{}^2(1-\cos\theta)+\cos\theta\end{bmatrix}$$

8.3 Scale

We can scale an object to make it proportionally bigger or smaller by a factor of k. If we apply the same scale the same in every direction, "dilating" the object about the origin, we are performing a *uniform* scale. Uniform scale preserves angles and proportions. Lengths increase or decrease uniformly by a factor of k, areas by a factor of k^2, and volumes (in 3D) by a factor of k^3.

If we wish to "stretch" or "squash" the object, we can apply different scale factors in different directions, resulting in *non-uniform* scale. Non-uniform scale does not preserve angles. Lengths, areas, and volumes are adjusted by a factor that varies according to the orientation relative to the direction of scale.

If $|k| < 1$, then the object gets "shorter" in that direction. If $|k| > 1$, then the object gets "longer." If $k = 0$, then we have an *orthographic projection*. We will discuss orthographic projection in Section 8.4. If $k < 0$, then we have a *reflection*. Reflections are covered in Section 8.5. For the remainder of this section, we will assume that $k > 0$.

Applying non-uniform scale has an effect very similar to shearing (see Section 8.6). In fact, it is impossible to distinguish between shearing and non-uniform scale.

8.3.1 Scaling along Cardinal Axes

The simplest way to perform scale is to apply a separate scale factor along each cardinal axis. The scale is applied about the perpendicular axis (in 2D) or plane (in 3D). If the scale factors for all axes are equal, then the scale is uniform; otherwise, it is non-uniform.

In 2D, we have two scale factors, k_x and k_y. Figure 8.13 shows an object with various scale values for k_x and k_y.

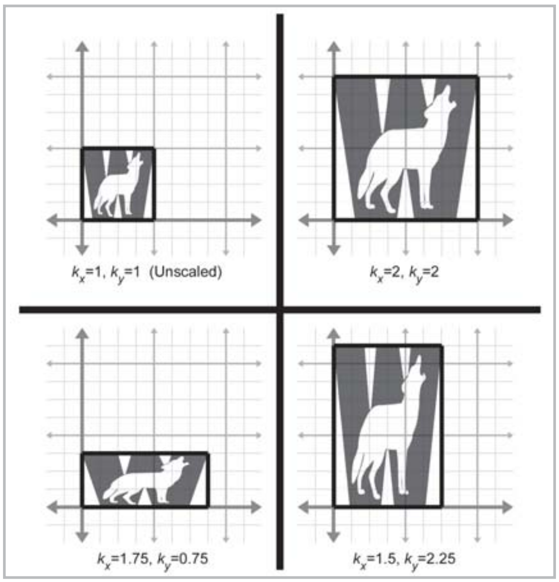


Figure 8.13: Scaling a 2D object with various factors for k_x and k_y

As is intuitively obvious, the basis vectors **p** and **q** are independently affected by the corresponding scale factors:

$$\begin{aligned} \mathbf{p}' &= k_x\mathbf{p} &= k_x\begin{bmatrix} 1 & 0 \end{bmatrix} &= \begin{bmatrix} k_x & 0 \end{bmatrix} \\ \mathbf{q}' &= k_y\mathbf{q} &= k_y\begin{bmatrix} 0 & 1 \end{bmatrix} &= \begin{bmatrix} 0 & k_y \end{bmatrix} \end{aligned}$$

Constructing the matrix from the basis vectors:

Equation 8.6: 2D matrix to scale on cardinal axes

$$\mathbf{S}(k_x, k_y) = \begin{bmatrix} \mathbf{p}' \\ \mathbf{q}' \end{bmatrix} = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix}$$

For 3D, we add a third scale factor k_z, and the 3D scale matrix is then given by:

Equation 8.7: 3D matrix to scale on cardinal axes

$$\mathbf{S}(k_x, k_y, k_z) = \begin{bmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{bmatrix}$$

8.3.2 Scale in an Arbitrary Direction

We can apply scale independent of the coordinate system used by scaling in an arbitrary direction. We will define **n** to be the unit vector parallel to the direction of scale, and *k* will be the scale factor to be applied about the line (in 2D) or plane (in 3D) that passes through the origin and is perpendicular to **n**.

Let's derive an expression that, given an arbitrary vector **v**, computes **v**' in terms of **v**, **n**, and *k*. To do this, we will separate **v** into two values, $\mathbf{v}_{\parallel}$ and $\mathbf{v}_{\perp}$, which are parallel and perpendicular to **n**, respectively, such that $\mathbf{v} = \mathbf{v}_{\parallel} + \mathbf{v}_{\perp}$. $\mathbf{v}_{\parallel}$ is the projection of **v** onto **n**. From Section 5.10.3, we know that $\mathbf{v}_{\parallel}$ is given by $(\mathbf{v}\cdot\mathbf{n})\mathbf{n}$. Since $\mathbf{v}_{\perp}$ is perpendicular to **n**, it will not be affected by the scale operation. Thus, $\mathbf{v}'=\mathbf{v}'_{\parallel} + \mathbf{v}_{\perp}$, and all we are left with is to compute the value of $\mathbf{v}'_{\parallel}$. Since $\mathbf{v}_{\parallel}$ is parallel to the direction of scale, $\mathbf{v}'_{\parallel}$ is trivially given by $k\mathbf{v}_{\parallel}$. This is shown below in Figure 8.14:

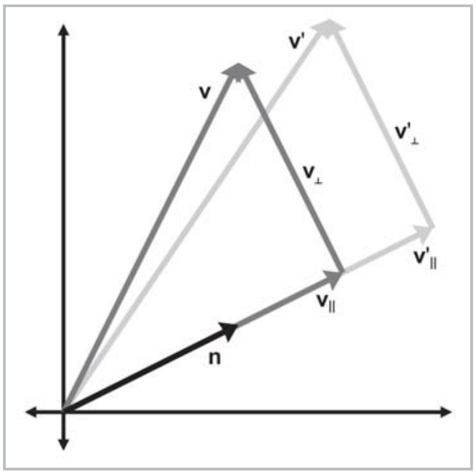


Figure 8.14: Scaling a vector along an arbitrary direction

Summarizing the known vectors and substituting, we have:

$$\begin{aligned}
\mathbf{v} &= \mathbf{v}_{\|} + \mathbf{v}_{\perp} \\
\mathbf{v}_{\|} &= (\mathbf{v} \cdot \mathbf{n})\, \mathbf{n} \\
\mathbf{v}'_{\perp} &= \mathbf{v}_{\perp} \\
&= \mathbf{v} - \mathbf{v}_{\|} \\
&= \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\, \mathbf{n} \\
\mathbf{v}'_{\|} &= k\mathbf{v}_{\|} \\
&= k\,(\mathbf{v} \cdot \mathbf{n})\, \mathbf{n} \\
\mathbf{v}' &= \mathbf{v}'_{\perp} + \mathbf{v}'_{\|} \\
&= \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\, \mathbf{n} + k\,(\mathbf{v} \cdot \mathbf{n})\, \mathbf{n} \\
&= \mathbf{v} + (k-1)\,(\mathbf{v} \cdot \mathbf{n})\, \mathbf{n}
\end{aligned}$$

Now that we know how to scale an arbitrary vector, we can compute the value of the basis vectors after scaling. We'll work through the first basis vector in 2D. The other basis vector is derived in a similar manner, and so we merely present the results. (Note that column vectors are used in the equations below strictly to make the equations format nicely.)

$$\begin{aligned}
\mathbf{p} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \\
\mathbf{p}' &= \mathbf{p} + (k-1)\,(\mathbf{p} \cdot \mathbf{n})\, \mathbf{n} \\
&= \begin{bmatrix} 1 \\ 0 \end{bmatrix} + (k-1) \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{n}_x \\ \mathbf{n}_y \end{bmatrix} \right) \begin{bmatrix} \mathbf{n}_x \\ \mathbf{n}_y \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ 0 \end{bmatrix} + (k-1)\, \mathbf{n}_x \begin{bmatrix} \mathbf{n}_x \\ \mathbf{n}_y \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} (k-1)\, {\mathbf{n}_x}^2 \\ (k-1)\, \mathbf{n}_x \mathbf{n}_y \end{bmatrix} \\
&= \begin{bmatrix} 1 + (k-1)\, {\mathbf{n}_x}^2 \\ (k-1)\, \mathbf{n}_x \mathbf{n}_y \end{bmatrix} \\
\mathbf{q} &= \begin{bmatrix} 0 & 1 \end{bmatrix} \\
\mathbf{q}' &= \begin{bmatrix} (k-1)\, \mathbf{n}_x \mathbf{n}_y \\ 1 + (k-1)\, {\mathbf{n}_y}^2 \end{bmatrix}
\end{aligned}$$

Forming a matrix from the basis vectors, we arrive at the 2D matrix to scale by a factor of *k* in an arbitrary direction specified by the unit vector **n**:

Equation 8.8: 2D matrix to scale in an arbitrary direction

$$\mathbf{S}(\mathbf{n}, k) = \begin{bmatrix} \mathbf{p}' \\ \mathbf{q}' \end{bmatrix} \begin{bmatrix} 1 + (k-1)\, {\mathbf{n}_x}^2 & (k-1)\, \mathbf{n}_x \mathbf{n}_y \\ (k-1)\, \mathbf{n}_x \mathbf{n}_y & 1 + (k-1)\, {\mathbf{n}_y}^2 \end{bmatrix}$$

In 3D, the basis vectors are computed by:

$$
\begin{aligned}
\mathbf{p} &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \\
\mathbf{p}' &= \mathbf{p} + (k-1)(\mathbf{p}\cdot\mathbf{n})\,\mathbf{n} \\
&= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + (k-1)\left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{n}_x \\ \mathbf{n}_y \\ \mathbf{n}_z \end{bmatrix}\right)\begin{bmatrix} \mathbf{n}_x \\ \mathbf{n}_y \\ \mathbf{n}_z \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + (k-1)\,\mathbf{n}_x \begin{bmatrix} \mathbf{n}_x \\ \mathbf{n}_y \\ \mathbf{n}_z \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} (k-1)\,{\mathbf{n}_x}^2 \\ (k-1)\,\mathbf{n}_x\mathbf{n}_y \\ (k-1)\,\mathbf{n}_x\mathbf{n}_z \end{bmatrix} \\
&= \begin{bmatrix} 1+(k-1)\,{\mathbf{n}_x}^2 \\ (k-1)\,\mathbf{n}_x\mathbf{n}_y \\ (k-1)\,\mathbf{n}_x\mathbf{n}_z \end{bmatrix} \\
\mathbf{q} &= \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \\
\mathbf{q}' &= \begin{bmatrix} (k-1)\,\mathbf{n}_x\mathbf{n}_y \\ 1+(k-1)\,{\mathbf{n}_y}^2 \\ (k-1)\,\mathbf{n}_y\mathbf{n}_z \end{bmatrix} \\
\mathbf{r} &= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \\
\mathbf{r}' &= \begin{bmatrix} (k-1)\,\mathbf{n}_x\mathbf{n}_z \\ (k-1)\,\mathbf{n}_y\mathbf{n}_z \\ 1+(k-1)\,{\mathbf{n}_z}^2 \end{bmatrix}
\end{aligned}
$$

The 3D matrix to scale by a factor of k in an arbitrary direction specified by the unit vector $\mathbf{n}$ is:

Equation 8.9: 3D matrix to scale in an arbitrary direction

$$
\mathbf{S}(\mathbf{n},k) = \begin{bmatrix} \mathbf{p}' \\ \mathbf{q}' \\ \mathbf{r}' \end{bmatrix} \begin{bmatrix} 1+(k-1)\,{\mathbf{n}_x}^2 & (k-1)\,\mathbf{n}_x\mathbf{n}_y & (k-1)\,\mathbf{n}_x\mathbf{n}_z \\ (k-1)\,\mathbf{n}_x\mathbf{n}_y & 1+(k-1)\,{\mathbf{n}_y}^2 & (k-1)\,\mathbf{n}_y\mathbf{n}_z \\ (k-1)\,\mathbf{n}_x\mathbf{n}_z & (k-1)\,\mathbf{n}_y\mathbf{n}_z & 1+(k-1)\,{\mathbf{n}_z}^2 \end{bmatrix}
$$

8.4 Orthographic Projection

In general, the term *projection* refers to any dimension-reducing operation. As we mentioned in Section 8.3, one way we can achieve projection is to use a scale factor of zero in a direction. In this case, all the points are flattened, or *projected*, onto the perpendicular axis (in 2D) or plane (in 3D). This type of projection is an *orthographic* projection, also known as a *parallel* projection, since the lines from the original points to their projected counterparts are parallel. We will learn about another type of projection, perspective projection, in Section 9.4.4.

8.4.1 Projecting onto a Cardinal Axis or Plane

The simplest type of projection occurs when we project onto a cardinal axis (in 2D) or plane (in 3D). This is illustrated in Figure 8.15:

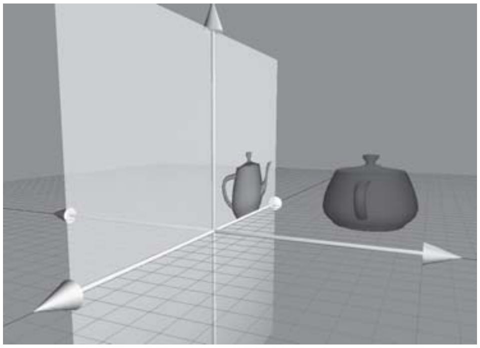


Figure 8.15: Projecting a 3D object onto a cardinal plane

Projection onto a cardinal axis or plane most frequently occurs not by actual transformation, but by simply discarding one of the dimensions while assigning the data into a variable of lesser dimension. For example, we may turn a 3D object into a 2D object by discarding the *z* components of the points and copying only *x* and *y*.

However, we can also project onto a cardinal axis or plane by using a scale value of zero on the perpendicular axis. For completeness, we will present the matrices for these transformations:

Equation 8.10: 2D matrix to project onto the *x*-axis

$$\mathbf{P}_x = \mathbf{S}\left(\begin{bmatrix} 0 & 1 \end{bmatrix}, 0\right) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Equation 8.11: 2D matrix to project onto the *y*-axis

$$\mathbf{P}_y = \mathbf{S}\left(\begin{bmatrix} 1 & 0 \end{bmatrix}, 0\right) = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Equation 8.12: 3D matrix to project onto the *xy*-plane

$$\mathbf{P}_{xy} = \mathbf{S}\left(\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}, 0\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Equation 8.13: 3D matrix to project onto the *xz*-plane

$$\mathbf{P}_{xz} = \mathbf{S}\left(\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, 0\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Equation 8.14: 3D matrix to project onto the *yz*-plane

$$\mathbf{P}_{yz} = \mathbf{S}\left(\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, 0\right) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

8.4.2 Projecting onto an Arbitrary Line or Plane

We can also project onto any arbitrary line (in 2D) or plane (in 3D). As always, since we are not considering translation, the line or plane must pass through the origin. The projection will be defined by a unit vector **n** that is perpendicular to the line or plane.

We can derive the matrix to project in an arbitrary direction by applying a zero scale factor along this direction, using the equations we developed in 8.3.2. In 2D:

Equation 8.15: 2D matrix to project onto an arbitrary line

$$\begin{aligned}\mathbf{P}(\mathbf{n}) &= \mathbf{S}(\mathbf{n},0)\\ &= \begin{bmatrix} 1+(0-1)\,\mathbf{n}_x{}^2 & (0-1)\,\mathbf{n}_x\mathbf{n}_y \\ (0-1)\,\mathbf{n}_x\mathbf{n}_y & 1+(0-1)\,\mathbf{n}_y{}^2 \end{bmatrix}\\ &= \begin{bmatrix} 1-\mathbf{n}_x{}^2 & -\mathbf{n}_x\mathbf{n}_y \\ -\mathbf{n}_x\mathbf{n}_y & 1-\mathbf{n}_y{}^2 \end{bmatrix}\end{aligned}$$

Remember that **n** is *perpendicular* to the line onto which we are projecting, not parallel to it. In 3D, we project onto the plane perpendicular to **n**:

Equation 8.16: 3D matrix to project onto an arbitrary plane

$$\begin{aligned}\mathbf{P}(\mathbf{n}) &= \mathbf{S}(\mathbf{n},0)\\ &= \begin{bmatrix} 1+(0-1)\,\mathbf{n}_x{}^2 & (0-1)\,\mathbf{n}_x\mathbf{n}_y & (0-1)\,\mathbf{n}_x\mathbf{n}_z \\ (0-1)\,\mathbf{n}_x\mathbf{n}_y & 1+(0-1)\,\mathbf{n}_y{}^2 & (0-1)\,\mathbf{n}_y\mathbf{n}_z \\ (0-1)\,\mathbf{n}_x\mathbf{n}_z & (0-1)\,\mathbf{n}_y\mathbf{n}_z & 1+(0-1)\,\mathbf{n}_z{}^2 \end{bmatrix}\\ &= \begin{bmatrix} 1-\mathbf{n}_x{}^2 & -\mathbf{n}_x\mathbf{n}_y & -\mathbf{n}_x\mathbf{n}_z \\ -\mathbf{n}_x\mathbf{n}_y & 1-\mathbf{n}_y{}^2 & -\mathbf{n}_y\mathbf{n}_z \\ -\mathbf{n}_x\mathbf{n}_z & -\mathbf{n}_y\mathbf{n}_z & 1-\mathbf{n}_z{}^2 \end{bmatrix}\end{aligned}$$

8.5 Reflection

Reflection (also called *mirroring*) is a transformation that "flips" the object about a line (in 2D) or a plane (in 3D). Figure 8.16 shows the result of reflecting an object.

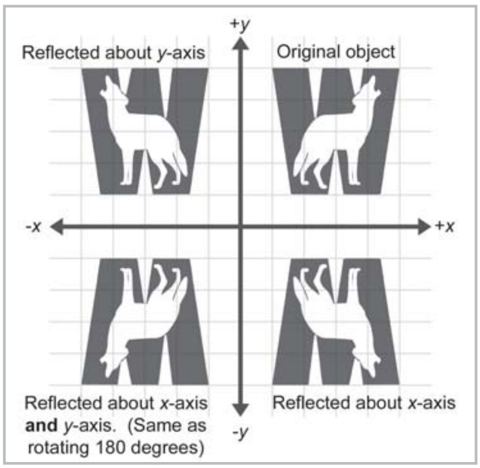


Figure 8.16: Reflecting an object about an axis in 2D

Reflection can be accomplished easily by applying a scale factor of –1. Let **n** be a 2D unit vector. Then the following matrix performs a reflection about the axis of reflection that passes through the origin and is perpendicular to **n**:

Equation 8.17: 2D matrix to reflect about an arbitrary axis

$$\begin{aligned}\mathbf{R}(\mathbf{n}) &= \mathbf{S}(\mathbf{n}, -1) \\ &= \begin{bmatrix} 1+(-1-1)\,{\mathbf{n}_x}^2 & (-1-1)\,\mathbf{n}_x\mathbf{n}_y \\ (-1-1)\,\mathbf{n}_x\mathbf{n}_y & 1+(-1-1)\,{\mathbf{n}_y}^2 \end{bmatrix} \\ &= \begin{bmatrix} 1-2{\mathbf{n}_x}^2 & -2\mathbf{n}_x\mathbf{n}_y \\ -2\mathbf{n}_x\mathbf{n}_y & 1-2{\mathbf{n}_y}^2 \end{bmatrix}\end{aligned}$$

In 3D, we have a reflecting plane instead of an axis. The following matrix reflects about a plane through the origin perpendicular to the unit vector **n**:

Equation 8.18: 3D matrix to reflect about an arbitrary plane

$$\begin{aligned}\mathbf{R}(\mathbf{n}) &= \mathbf{S}(\mathbf{n}, -1) \\ &= \begin{bmatrix} 1+(-1-1)\,{\mathbf{n}_x}^2 & (-1-1)\,\mathbf{n}_x\mathbf{n}_y & (-1-1)\,\mathbf{n}_x\mathbf{n}_z \\ (-1-1)\,\mathbf{n}_x\mathbf{n}_y & 1+(-1-1)\,{\mathbf{n}_y}^2 & (-1-1)\,\mathbf{n}_y\mathbf{n}_z \\ (-1-1)\,\mathbf{n}_x\mathbf{n}_z & (-1-1)\,\mathbf{n}_y\mathbf{n}_z & 1+(-1-1)\,{\mathbf{n}_z}^2 \end{bmatrix} \\ &= \begin{bmatrix} 1-2{\mathbf{n}_x}^2 & -2\mathbf{n}_x\mathbf{n}_y & -2\mathbf{n}_x\mathbf{n}_z \\ -2\mathbf{n}_x\mathbf{n}_y & 1-2{\mathbf{n}_y}^2 & -2\mathbf{n}_y\mathbf{n}_z \\ -2\mathbf{n}_x\mathbf{n}_z & -2\mathbf{n}_y\mathbf{n}_z & 1-2{\mathbf{n}_z}^2 \end{bmatrix}\end{aligned}$$

Notice that an object can only be "reflected" once. If we reflect it again (even about a different axis or plane), then the object is flipped back to "right side out," and it is the same as if we had rotated the object from its initial position.

8.6 Shearing

Shearing is a transformation that "skews" the coordinate space, stretching it non-uniformly. Angles are not preserved; however, surprisingly, areas and volumes are. The basic idea is to add a multiple of one coordinate to the other. For example, in 2D, we might take a multiple of y and add it to x, so that $x' = x + sy$. This is shown in Figure 8.17:

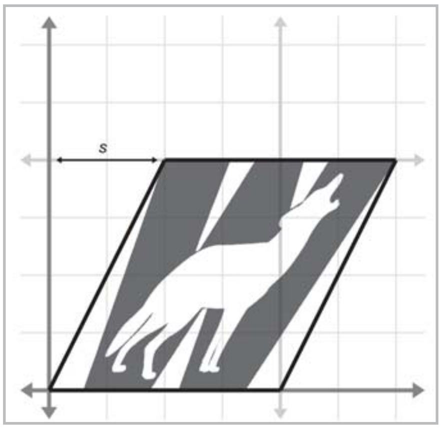


Figure 8.17: Shearing in 2D

The matrix that performs this shear is:

$$\mathbf{H}_x(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

The notation $\mathbf{H}_x$ denotes that the x coordinate is sheared by the other coordinate, y. The parameter s controls the amount and direction of the shearing. The other 2D shear matrix, $\mathbf{H}_y$ is given below:

$$\mathbf{H}_y(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

In 3D, we can take one coordinate and add different multiples of that coordinate to the other two coordinates. The notation $\mathbf{H}_{xy}$ indicates that the x and y coordinates are shifted by the other coordinate, z. These matrices are given below:

Equation 8.19: 3D shear matrices

$$\begin{aligned}
\mathbf{H}_{xy}(s,t) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ s & t & 1 \end{bmatrix} \\
\mathbf{H}_{xz}(s,t) &= \begin{bmatrix} 1 & 0 & 0 \\ s & 1 & t \\ 0 & 0 & 1 \end{bmatrix} \\
\mathbf{H}_{yz}(s,t) &= \begin{bmatrix} 1 & s & t \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

Shearing is a seldom-used transform. It is also known as a *skew* transform. Combining shearing and scaling (uniform or non-uniform) creates a transformation that is indistinguishable from a transformation containing rotation and non-uniform scale.

8.7 Combining Transformations

In this section we show how to take a sequence of transformation matrices and combine (or "concatenate") them into one single transformation matrix. This new matrix will represent the cumulative result of applying all of the original transformations in order.

One very common example of this is in rendering. Imagine there is an object at an arbitrary position and orientation in the world. We wish to render this object given a camera in any position and orientation. To do this, we must take the vertices of the object (assuming we are rendering some sort of triangle mesh) and transform them from object space into world space, and then from world space into camera space. The math involved is summarized below:

$$\begin{aligned}
\mathbf{p}_{world} &= \mathbf{p}_{object}\mathbf{M}_{object\rightarrow world} \\
\mathbf{p}_{camera} &= \mathbf{p}_{world}\mathbf{M}_{world\rightarrow camera} \\
&= (\mathbf{p}_{object}\mathbf{M}_{object\rightarrow world})\mathbf{M}_{world\rightarrow camera}
\end{aligned}$$

From Section 7.1.6 we know that matrix multiplication is associative, and so we can compute one matrix to transform directly from object to camera space:

$$\begin{aligned}\mathbf{p}_{camera} &= (\mathbf{p}_{object}\mathbf{M}_{object\rightarrow world})\mathbf{M}_{world\rightarrow camera} \\ &= \mathbf{p}_{object}(\mathbf{M}_{object\rightarrow world}\mathbf{M}_{world\rightarrow camera})\end{aligned}$$

Thus, we can concatenate the matrices outside the loop and have only one matrix multiplication inside the loop (remember there are many vertices):

$$\begin{aligned}\mathbf{M}_{object\rightarrow camera} &= \mathbf{M}_{object\rightarrow world}\mathbf{M}_{world\rightarrow camera} \\ \mathbf{p}_{camera} &= \mathbf{p}_{object}\mathbf{M}_{object\rightarrow camera}\end{aligned}$$

So we see that matrix concatenation works from an algebraic perspective using the associative property of matrix multiplication. Let's see if we can't get a more geometric interpretation of what's going on. Recall our breakthrough discovery from Section 7.2 that the rows of a matrix contain the basis vectors after transformation. This is true even in the case of multiple transformations. Notice that in the matrix product **AB**, each resulting row is the product of the corresponding row from **A** times the matrix **B**. In other words, let the row vectors $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{a}_3$ stand for the rows of **A**. Then matrix multiplication can alternatively be written like this:

$$\begin{aligned}\mathbf{A} &= \begin{bmatrix}\mathbf{a}_1\\ \mathbf{a}_2\\ \mathbf{a}_3\end{bmatrix} \\ \mathbf{AB} &= \begin{bmatrix}\mathbf{a}_1\\ \mathbf{a}_2\\ \mathbf{a}_3\end{bmatrix}\mathbf{B} \\ &= \begin{bmatrix}\mathbf{a}_1\mathbf{B}\\ \mathbf{a}_2\mathbf{B}\\ \mathbf{a}_3\mathbf{B}\end{bmatrix}\end{aligned}$$

This makes it explicitly clear that the rows of the product of **AB** are actually the result of transforming the basis vectors in **A** by **B**.

8.8 Classes of Transformations

We can classify transformations using several criteria. In this section, we will discuss classes of transformations. For each class, we will describe the properties of the transformations that belong to that class and specify which of the primitive transformations from Sections 8.2 through 8.6 belong to that class.

The classes of transformations are not mutually exclusive, nor do they necessarily follow an "order" or "hierarchy" with each one more or less restrictive than the next.

When we discuss transformations in general, we may make use of the synonymous terms *mapping* or *function*. In the most general sense, a *mapping* is simply a rule that takes an input and produces an output. We denote that a mapping **F** maps **a** to **b** by writing **F**(**a**) = **b** (read "**F** of **a**

equals **b**”). Of course, we will be primarily interested in mappings that can be expressed using matrix multiplication, but it is important to note that other mappings are possible.

8.8.1 Linear Transformations

We met linear functions informally in Section 7.2. Mathematically, a mapping **F**(**a**) is *linear* if

$$\mathbf{F}(\mathbf{a}+\mathbf{b}) = \mathbf{F}(\mathbf{a}) + \mathbf{F}(\mathbf{b})$$

and

$$\mathbf{F}(k\mathbf{a}) = k\mathbf{F}(\mathbf{a})$$

This is a fancy way of stating that the mapping **F** is linear if it preserves the basic operations of addition and multiplication by a scalar. If we add two vectors and then perform the transformation, we get the same result as if we perform the transformation on the two vectors individually and then add the transformed vectors. Likewise, if we scale a vector and then transform it, we should get the same resulting vector as when we transform the vector and then scale it.

There are two important implications of this definition of linear transformation:

- The mapping **F**(**a**) = **aM**, where **M** is any square matrix, is a linear transformation because

  $$\begin{aligned}\mathbf{F}(\mathbf{a}+\mathbf{b}) &= (\mathbf{a}+\mathbf{b})\mathbf{M} \\ &= \mathbf{aM}+\mathbf{bM} \\ &= \mathbf{F}(\mathbf{a})+\mathbf{F}(\mathbf{b})\end{aligned}$$

  and

  $$\begin{aligned}\mathbf{F}(k\mathbf{a}) &= (k\mathbf{a})\mathbf{M} \\ &= k(\mathbf{aM}) \\ &= k\mathbf{F}(\mathbf{a})\end{aligned}$$

- Any linear transformation will transform the zero vector into the zero vector. (If **F**(**0**) = **a**, **a** ≠ **0**, then **F** cannot be a linear mapping, since **F**(*k***0**) = **a** and therefore **F**(*k***0**) ≠ *k***F**(**0**).) Because of this, linear transformations do not contain translation.

Since all of the transformations we discussed in Sections 8.2 through 8.6 can be expressed using matrix multiplication, they are all linear transformations.

In some literature, a linear transformation is defined as one in which parallel lines remain parallel after transformation. This is almost completely accurate, with one slight exception: projection. (When a line is projected and becomes a single point, can we consider that point parallel to anything?) Excluding this one technicality, the intuition is correct. A linear transformation may “stretch” things, but straight lines are not “warped” and parallel lines remain parallel.

8.8.2 Affine Transformations

An *affine* transformation is a linear transformation followed by translation. Thus, the set of affine transformations is a superset of the set of linear transformations. Any linear transformation is an affine translation, but not all affine transformations are linear transformations.

Since all of the transformations we discussed in this chapter are linear transformations, they are also all affine transformations.

The class of affine transformations is the most general class of transformations that we will consider. Any transformation of the form $\mathbf{v}' = \mathbf{vM} + \mathbf{b}$ is an affine transformation.

8.8.3 Invertible Transformations

A transformation is *invertible* if there exists an opposite transformation that "undoes" the original transformation. In other words, a mapping $\mathbf{F}(\mathbf{a})$ is invertible if there exists a mapping $\mathbf{F}^{-1}$, such that $\mathbf{F}^{-1}(\mathbf{F}(\mathbf{a})) = \mathbf{a}$ for all $\mathbf{a}$.

There are non-affine invertible transformations, but we will not consider them for the moment. For now we'll concentrate on determining if an affine transformation is invertible. An affine transformation is a linear transformation followed by a translation. Obviously, we can always "undo" the translation portion by simply translating by the opposite amount. So the question becomes whether or not the linear transformation is invertible.

Intuitively, we know that all of the transformations other than projection can be "undone." When an object is projected, we effectively discard one dimension worth of information, and this information cannot be recovered. Thus, all of the primitive transformations other than projection are invertible.

Since any linear transformation can be expressed as multiplication by a matrix, finding the inverse of a linear transformation is equivalent to finding the inverse of a matrix. We will discuss how to do this in Section 9.2. If the matrix is *singular* (it has no inverse), then the transformation is *non-invertible*. The determinant of an invertible matrix is nonzero.

8.8.4 Angle-preserving Transformations

A transformation is *angle-preserving* if the angle between two vectors is not altered in either magnitude or direction after transformation. Only translation, rotation, and uniform scale are angle-preserving transformations. An angle-preserving matrix preserves proportions. We do not consider reflection an angle-preserving transformation because even though the *amount* of angle between two vectors is the same after transformation, the *direction* of angle may be inverted. All angle-preserving transformations are affine and invertible.

8.8.5 Orthogonal Transformations

Orthogonal is a term that is used to describe a matrix with certain properties. We will defer a complete discussion of orthogonal matrices until Section 9.3, but the basic idea is that the axes remain perpendicular, and no scale is applied. Orthogonal transformations are interesting because it is easy to compute their inverse.

Translation, rotation, and reflection are the only orthogonal transformations. Lengths, angles, areas, and volumes are all preserved (although in saying this we must be careful of our precise definition of angle, area, and volume, since reflection is an orthogonal transformation).

As we will learn in Chapter 9, the determinant of an orthogonal matrix is ±1.

All orthogonal transformations are affine and invertible.

8.8.6 Rigid Body Transformations

A *rigid body* transformation is one that changes the location and orientation of an object but not its shape. All angles, lengths, areas, and volumes are preserved. Translation and rotation are the only rigid body transformations. Reflection is not considered a rigid body transformation.

Rigid body transformations are also known as *proper* transformations. All rigid body transformations are orthogonal, angle-preserving, invertible, and affine.

The determinant of any rotation matrix is 1.

8.8.7 Summary of Types of Transformations

The following table summarizes the relationship between the various classes and types of transformations. In this table, a "yes" means that the transformation in that row *always* has the property associated with that column. The absence of a "yes" does not mean "never," but rather, "not always."

Transform	Linear	Affine	Invertible	Angle preserving	Orthogonal	Rigid Body	Lengths preserved	Areas/ volumes preserved	Determinant
Linear transformations	Y	Y							
Affine transformations		Y							
Invertible transformations			Y						0
Angle-preserving transformations		Y	Y	Y					
Orthogonal transformations		Y	Y		Y				1
Rigid body transformations		Y	Y	Y	Y	Y	Y	Y	
Translation		Y	Y	Y	Y	Y	Y	Y	
Rotation[1]	Y	Y	Y	Y	Y	Y	Y	Y	1
Uniform scale[2]	Y	Y	Y	Y					K^n [3]
Non-uniform scale[2]	Y	Y	Y						
Orthographic projection[4]	Y	Y							0